Browser Use Cloud — Full Documentation

Quick start
Source: https://docs.browser-use.com/cloud/quickstart

1. Install

` ` ` bash Python
pip install browser-use-sdk

` ` `

` ` ` bash TypeScript
npm install browser-use-sdk

` ` `

Get a key at
[cloud.browser-use.com/settings](https://cloud.browser-use.com/settings?tab=api-keys&new=1), then:

` ` ` bash
export BROWSER_USE_API_KEY=your_key

` ` `

2. Run your first task

` ` ` python Python
import asyncio
from browser_use_sdk.v3 import AsyncBrowserUse

async def main():
client = AsyncBrowserUse()
result = await client.run("List the top 20 posts on Hacker News today with their points")
print(result.output)

asyncio.run(main())

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();

const result = await client.run("List the top 20 posts on Hacker News today with their points"); console.log(result.output);

` ` `

Want a full working app? Check out the [Chat UI example] (https://docs.browser-use.com/cloud/tutorials/chat-ui).

Agent vs Browser

	Agent	**Browser**
Method	`sessions.create()` / `run()`
` browsers.create()`
What it does	AI agent runs your task	Raw browser via CDP
task	✓	—
model	✓	—
proxy	✓	✓
custom_proxy	✓	✓
profile_id	✓	✓
recording	✓	✓
workspace_id	✓	—
keep_alive	✓	—
screen size	—	✓
timeout	—	✓

- - -

If you are an LLM, read/include docs.browser-use.com/llms-full.txt — it contains the complete SDK reference with all code examples in a single file optimized for LLMs. For a shorter index: docs.browser-use.com/llms.txt.

Prompt for Vibecoders
Source: https://docs.browser-use.com/cloud/vibecoding

Copy this link and paste it into your coding agent (Cursor,

Claude Code, Windsurf, etc.) — it contains all the context needed to build with Browser Use.

` ` `

https://docs.browser-use.com/cloud/llms.txt

` ` `

Introduction
Source: https://docs.browser-use.com/cloud/agent/quickstart

The SDK is a thin wrapper around the [API v3 Reference] (https://docs.browser-use.com/cloud/api-reference). Every endpoint in the API reference is available as an SDK method — `client.sessions`, `client.browsers`, `client.profiles`,
` client.workspaces`, and `client.billing`.

` client.run()` creates a session, polls every 2 seconds
until completion (up to 4 hours), and returns the result. It accepts all parameters from the [Create Session] (https://docs.browser-use.com/cloud/api-v3/sessions/create-session) endpoint. The result is a [Session object] (https://docs.browser-use.com/cloud/api-v3/sessions/get-session) — use `result.output` for the agent's response.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
result = await client.run("List the top 20 posts on Hacker News today with their points")
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const result = await client.run("List the top 20 posts on Hacker News today with their points"); console.log(result.output);

` ` `

` ` ` bash curl
curl -X POST https://api.browser-use.com/api/v3/sessions \
- H "X-Browser-Use-API-Key: YOUR_API_KEY" \
- H "Content-Type: application/json" \
- d '{"task": "List the top 20 posts on Hacker News today
with their points"}'

` ` `

What this agent can do:
- **Data extraction** — scrape websites with thousands of listings
- **Form filling** — submit applications, fill out surveys, enter data
- **Multi-step workflows** — log in, navigate, click through flows, download files
- **Research** — search across multiple sites, compare results, summarize findings
- **Monitoring** — monitor a website and get notified if something changes
- **Testing** — test websites end-to-end with natural language instructions
- **Scheduling** — schedule tasks to run on a recurring basis
- **1,000+ integrations** — Gmail, Calendar, Notion, and
more

The best SOTA browser agent — see our [online Mind2Web benchmark](https://browser-use.com/posts/online-mind2web-benchmark).

If you are an LLM, read/include docs.browser-use.com/llms-full.txt — it contains the complete SDK reference with all code examples in a single file optimized for LLMs.

Structured output
Source:
https://docs.browser-use.com/cloud/agent/structured-output

Pass a Pydantic model (Python) or Zod schema (TypeScript) —

` result.output` is automatically validated and converted to
the typed object.

TypeScript requires **Zod v4** (`npm install zod@4`). Zod v3 is not compatible.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse
from pydantic import BaseModel

class Post(BaseModel):
name: str

points: int
comments: int

class HNPosts(BaseModel):
posts: list[Post]

client = AsyncBrowserUse()
result = await client.run(
"List the top 20 posts on Hacker News today with their points",
output_schema=HNPosts,
)
for post in result.output.posts:
print(f"{post.name} ({post.points} pts, {post.comments} comments)")

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3"; import { z } from "zod";

const Post = z.object({
name: z.string(),
points: z.number(),
comments: z.number(),
});

const HNPosts = z.object({ posts: z.array(Post),
});

const client = new BrowserUse();
const result = await client.run(
"List the top 20 posts on Hacker News today with their points",
{ schema: HNPosts },
);
for (const post of result.output.posts) { console.log(`${post.name} (${post.points} pts, $ {post.comments} comments)`);
}

` ` `

Follow-up tasks
Source: https://docs.browser-use.com/cloud/agent/follow-up-tasks

When you pass a `session_id`, the session automatically stays alive between tasks. Each task runs a new agent that reuses the same browser — the agents don't share context, but the browser state (page, cookies, tabs) carries over.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

# Create a session, then run tasks inside it
session = await client.sessions.create()

result1 = await client.run(
"Go to amazon.com, search for laptops, and open the first result",
session_id=session.id,
)
result2 = await client.run(
"Extract the customer reviews",
session_id=session.id,
)

await client.sessions.stop(session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();

// Create a session, then run tasks inside it
const session = await client.sessions.create();

const result1 = await client.run("Go to amazon.com, search for laptops, and open the first result", {
sessionId: session.id,
});
const result2 = await client.run("Extract the customer reviews", {
sessionId: session.id,
});

await client.sessions.stop(session.id);

` ` `

` sessions.create()` returns a `live_url` you can embed to
watch each task execute — see [Live preview](https://docs.browser-use.com/cloud/browser/live-preview). To stream messages as each task runs, use
` client.run()` with `for await` — see [Live messages]
(https://docs.browser-use.com/cloud/agent/streaming).

Sessions time out after 15 minutes of inactivity by default. The maximum session duration is 4 hours.

Live messages
Source: https://docs.browser-use.com/cloud/agent/streaming

Want a ready-made UI? See the [Chat UI tutorial] (https://docs.browser-use.com/cloud/tutorials/chat-ui).

Stream messages as the agent works — reasoning, tool calls, browser actions, and results. Each message has `role`,
` type`, `summary`, `data`, and `screenshot_url`. See [List

session messages](https://docs.browser-use.com/cloud/api-v3/sessions/list-session-messages) for all fields.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

run = client.run("Find the top story on Hacker News") async for msg in run:
print(f"[{msg.role}] {msg.summary}")

print(run.result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();

const run = client.run("Find the top story on Hacker News");
for await (const msg of run) { console.log(`[${msg.role}] ${msg.summary}`);
}

console.log(run.result.output);

` ` `

` ` `

[user] Find the top story on Hacker News
[assistant] Navigating to https://news.ycombinator.com/[tool] Browser Navigate: Navigated
[assistant] Analyzing browser state
[tool] Browser Analyze State: The top story is "Coding Agents Could Make Free Software Matter Again"
[tool] Done Autonomous: The top story on Hacker News is "Coding Agents Could Make Free Software Matter Again"

` ` `

Manual polling

If you need full control over the polling loop (e.g. custom

interval, filtering):

` ` ` python Python
import asyncio
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
session = await client.sessions.create(task="Find the top story on Hacker News")

cursor = None
while True:
msgs = await client.sessions.messages(session.id, after=cursor, limit=100)
for m in msgs.messages:
print(f"[{m.role}] {m.summary}")
cursor = m.id

s = await client.sessions.get(session.id)
if s.status.value in ("idle", "stopped", "error", "timed_out"):
break
await asyncio.sleep(2)

print(s.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const session = await client.sessions.create({
task: "Find the top story on Hacker News",
});

let cursor: string | undefined;
while (true) {
const msgs = await client.sessions.messages(session.id, { after: cursor, limit: 100 });
for (const m of msgs.messages) {
console.log(`[${m.role}] ${m.summary}`);
cursor = m.id;
}

const s = await client.sessions.get(session.id); if (["idle", "stopped", "error", "timed_out"].includes(s.status)) { console.log(s.output);
break;
}
await new Promise((r) => setTimeout(r, 2000));
}

` ` `

Related

- [Live preview &
recording](https://docs.browser-use.com/cloud/browser/live-preview) — embed the browser alongside your message stream
- [Follow-up
tasks](https://docs.browser-use.com/cloud/agent/follow-up-tasks) — chain multiple tasks in one session while streaming each

Workspaces & files
Source: https://docs.browser-use.com/cloud/agent/workspaces

Workspaces give your agent persistent file storage. Two patterns cover almost every use case:

1. **You upload a file** → agent reads it
2. **Agent creates a file** → you download it

Upload a file

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
workspace = await client.workspaces.create(name="my-workspace")

Upload

await client.workspaces.upload(workspace.id, "people.csv")

# Agent can now read it
result = await client.run(
"Read people.csv and tell me who works at Google", workspace_id=workspace.id,
)
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const workspace = await client.workspaces.create({ name: "my-workspace" });

// Upload
await client.workspaces.upload(workspace.id, "people.csv");

// Agent can now read it
const result = await client.run(
"Read people.csv and tell me who works at Google",
{ workspaceId: workspace.id },
);
console.log(result.output);

` ` `

You can upload multiple files at once:

` ` ` python Python
await client.workspaces.upload(workspace.id, "data.csv", "config.json", "image.png")

` ` `

` ` ` typescript TypeScript
await client.workspaces.upload(workspace.id, "data.csv", "config.json", "image.png");

` ` `

Download files

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
workspace = await client.workspaces.create(name="my-workspace")

# Agent creates a file
result = await client.run(
"Go to Hacker News and save the top 3 posts as posts.json", workspace_id=workspace.id,
)

Download a single file
await client.workspaces.download(workspace.id, "posts.json", to="./posts.json")

# Or download everything
paths = await client.workspaces.download_all(workspace.id, to="./output")
for p in paths:
print(f"Downloaded: {p}")

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const workspace = await client.workspaces.create({ name: "my-workspace" });

// Agent creates a file
const result = await client.run(
"Go to Hacker News and save the top 3 posts as posts.json",
{ workspaceId: workspace.id },
);

// Download a single file
await client.workspaces.download(workspace.id, "posts.json", { to: "./posts.json" });

// Or download everything
const paths = await
client.workspaces.downloadAll(workspace.id, { to:

"./output" });
for (const p of paths) {
console.log(`Downloaded: ${p}`);
}

` ` `

Manage workspaces

` ` ` python Python
workspace = await client.workspaces.get(workspace_id) updated = await client.workspaces.update(workspace_id, name="renamed")
response = await client.workspaces.list()
for w in response.items:
print(w.id, w.name)
await client.workspaces.delete(workspace_id)

` ` `

` ` ` typescript TypeScript
const workspace = await client.workspaces.get(workspaceId); const updated = await client.workspaces.update(workspaceId, { name: "renamed" });
const response = await client.workspaces.list();
for (const w of response.items) {
console.log(w.id, w.name);
}
await client.workspaces.delete(workspaceId);

` ` `

Organize with prefixes

Use `prefix` to organize files into directories within a workspace:

` ` ` python Python
Upload into a subdirectory
await client.workspaces.upload(workspace.id, "report.pdf", prefix="reports/")

# List files in a subdirectory
files = await client.workspaces.files(workspace.id, prefix="reports/")
for f in files.files:

print(f.path, f.size)

Download only files from a subdirectory
await client.workspaces.download_all(workspace.id, to="./output", prefix="reports/")

` ` `

` ` ` typescript TypeScript
// Upload into a subdirectory
await client.workspaces.upload(workspace.id, "report.pdf", { prefix: "reports/" });

// List files in a subdirectory
const files = await client.workspaces.files(workspace.id, { prefix: "reports/" });
for (const f of files.files) {
console.log(f.path, f.size);
}

// Download only files from a subdirectory
await client.workspaces.downloadAll(workspace.id, { to: "./output", prefix: "reports/" });

` ` `

List and delete files

` ` ` python Python
# List all files
files = await client.workspaces.files(workspace.id) for f in files.files:
print(f.path, f.size)

Delete a single file
await client.workspaces.delete_file(workspace.id, "old-report.pdf")

# Check workspace storage usage
size = await client.workspaces.size(workspace.id) print(f"Used: {size}")

` ` `

` ` ` typescript TypeScript
// List all files
const files = await client.workspaces.files(workspace.id);

for (const f of files.files) {
console.log(f.path, f.size);
}

// Delete a single file
await client.workspaces.deleteFile(workspace.id, "old-report.pdf");

` ` `

Manage workspaces

You can also manage workspaces from [cloud.browser-use.com/settings](https://cloud.browser-use.com/settings? tab=workspaces).

Deleting a workspace permanently removes all its files. This cannot be undone.

Human in the loop
Source: https://docs.browser-use.com/cloud/agent/human-in-the-loop

Use cases
- Human enters payment info or approves a transaction, agent handles the rest
- Human navigates a complex auth flow, then hands back to agent
- Human reviews what the agent did before the agent continues

Flow

1. Create a session — it stays alive automatically when you pass `session_id` to `run()`
2. Run an agent task
3. Human interacts with the live browser
4. Send a new follow-up task

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

# 1. Create a session
session = await client.sessions.create() print(f"Live view: {session.live_url}")

# 2. Agent does the first part
result = await client.run(
"Go to amazon.com and search for noise cancelling headphones",
session_id=session.id,
)
print(result.output)

# 3. Human opens live_url and picks a product
input("Press Enter after you've selected a product in the live view...")

# 4. Agent continues where the human left off
result = await client.run(
"Get the details of the selected product — name, price, and rating",
session_id=session.id,
)
print(result.output)

Clean up
await client.sessions.stop(session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3"; import * as readline from "readline";

const client = new BrowserUse();

// 1. Create a session
const session = await client.sessions.create(); console.log(`Live view: ${session.liveUrl}`);

// 2. Agent does the first part
const searchResult = await client.run(

"Go to amazon.com and search for noise cancelling headphones",
{ sessionId: session.id },
);
console.log(searchResult.output);

// 3. Human opens liveUrl and picks a product
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
await new Promise((resolve) =>
rl.question("Press Enter after you've selected a product in the live view...", resolve),
);
rl.close();

// 4. Agent continues where the human left off
const result = await client.run(
"Get the details of the selected product — name, price, and rating",
{ sessionId: session.id },
);
console.log(result.output);

// Clean up
await client.sessions.stop(session.id);

` ` `

Introduction Stealth
Source: https://docs.browser-use.com/cloud/browser/stealth

See [how we perform in the hardest stealth benchmark] (https://browser-use.com/posts/stealth-benchmark).

What's included

Every cloud browser session runs in a hardened Chromium fork with stealth enabled by default — no configuration needed.

- **Anti-detect browser fingerprinting** — Canvas, WebGL, fonts, navigator, and other browser fingerprints are randomized per session to appear as a real user. Passes CreepJS, BrowserLeaks, and other fingerprint detectors.
- **Ad and cookie banner blocking** — Banners are dismissed automatically so the agent sees clean pages and executes faster.
- **Cloudflare / anti-bot bypass** — Works on sites protected by Cloudflare, PerimeterX, and other bot detection services.

Residential proxies

Residential proxies are enabled by default across 195+ countries. This makes browser sessions appear as real users from the target geography. See [Proxies](https://docs.browser-use.com/cloud/browser/proxie s) for details on geo-targeting and custom proxy configuration.

Proxies
Source: https://docs.browser-use.com/cloud/browser/proxies

A US residential proxy is active by default on every browser. To route through a different country, set
` proxy_country_code`. See the [API reference](https://docs.browser-use.com/cloud/api-v3/browse rs/create-browser-session) for all supported country codes.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
browser = await
client.browsers.create(proxy_country_code="de") print(browser.cdp_url) # ws://... print(browser.live_url) # debug view

With an agent:
result = await client.run("Get the price of iPhone 16 on

amazon.de", proxy_country_code="de")

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const browser = await
client.browsers.create({ proxyCountryCode: "de" }); console.log(browser.cdpUrl);
console.log(browser.liveUrl);

// With an agent:
// const result = await client.run("Get the price of iPhone 16 on amazon.de", { proxyCountryCode: "de" });

` ` `

Disable proxies

If your use case does not need proxies, for example QA testing.

` ` ` python Python
browser = await
client.browsers.create(proxy_country_code=None)

With an agent:
result = await client.run("Go to http://localhost:3000", proxy_country_code=None)

` ` `

` ` ` typescript TypeScript
const browser = await
client.browsers.create({ proxyCountryCode: null });

// With an agent:
// const result = await client.run("Go to http://localhost:3000", { proxyCountryCode: null });

` ` `

Custom proxy

Bring your own proxy server (HTTP or SOCKS5). Requires custom plan.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
browser = await client.browsers.create(custom_proxy={
"host": "proxy.example.com",
"port": 8080,
"username": "user",
"password": "pass",
},
)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const browser = await client.browsers.create({ customProxy: {
host: "proxy.example.com",
port: 8080,
username: "user",
password: "pass",
},
});

` ` `

Live preview & recording
Source: https://docs.browser-use.com/cloud/browser/live-preview

Want a ready-made UI? See the [Chat UI tutorial] (https://docs.browser-use.com/cloud/tutorials/chat-ui).

` liveUrl` is returned on session creation.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
session = await client.sessions.create(task="Check how many GitHub stars browser-use has")
print(session.live_url)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const session = await client.sessions.create({
task: "Check how many GitHub stars browser-use has", });
console.log(session.liveUrl);

` ` `

` liveUrl` is also returned when creating a standalone
browser session:

` ` ` python Python
browser = await client.browsers.create() print(browser.live_url)

` ` `

` ` ` typescript TypeScript
const browser = await client.browsers.create(); console.log(browser.liveUrl);

` ` `

Embed live browser into your app

Useful for human interaction or to see live what's happening.

` ` ` html
<iframe
src="{live_url}"
width="1280"
height="720"
allow="autoplay"
style="border: none; border-radius: 8px;" ></iframe>
` ` `

The live URL is hosted on `live.browser-use.com`. If your app sets a Content Security Policy, add it to your `frame-src` directive:

` ` `

Content-Security-Policy: frame-src 'self' https://live.browser-use.com;

` ` `

For responsive sizing, use CSS instead of fixed dimensions:

` ` ` html
<iframe
src="{live_url}"
style="width: 100%; aspect-ratio: 16/9; border: none; border-radius: 8px;"
allow="autoplay"
></iframe>
` ` `

Customize

Append query parameters to the `liveUrl`:

Parameter	Values	Description		-----------	--------	-------------
`theme`	`light`, `dark` (default)	Light or dark mode
`ui`	`false`	Hide the browser chrome (URL bar, tabs)

` ` `

https://live.browser-use.com?wss=...&theme=light https://live.browser-use.com?wss=...&ui=false

` ` `

Recording

`waitForRecording` / `wait_for_recording` requires the **v3 SDK** (`from browser_use_sdk.v3 import AsyncBrowserUse` / `import { BrowserUse } from "browser-use-sdk/v3"`).

Enable recording to get an MP4 video of the browser session. Only available when the agent actually opens a browser — tasks answered without browsing produce no recording. If you run multiple tasks in the same session (with `keep_alive`), you may get multiple recordings.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
result = await client.run(
"Check how many GitHub stars browser-use has", enable_recording=True,
)

# Waits up to 15s for recording to be ready. Returns [] if no browser was opened.
urls = await client.sessions.wait_for_recording(result.id) for url in urls:
print(url) # presigned MP4 download URL

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const result = await client.run("Check how many GitHub stars browser-use has", {
enableRecording: true,
});

// Waits up to 15s for recording to be ready. Returns [] if no browser was opened.
const urls = await
client.sessions.waitForRecording(result.id);
for (const url of urls) {
console.log(url); // presigned MP4 download URL
}

` ` `

For standalone browser sessions, pass `enable_recording` when creating the browser and retrieve the URL after

stopping it:

` ` ` python Python
browser = await
client.browsers.create(enable_recording=True)
# ... use the browser via CDP ...
stopped = await client.browsers.stop(browser.id) print(stopped.recording_url) # presigned MP4 download URL

` ` `

` ` ` typescript TypeScript
const browser = await
client.browsers.create({ enableRecording: true });
// ... use the browser via CDP ...
const stopped = await client.browsers.stop(browser.id); console.log(stopped.recordingUrl); // presigned MP4 download URL
` ` `

Recording URLs are presigned and **expire after 1 hour**. Download or serve the recording promptly. If you need to access it later, save the MP4 to your own storage.

Related

- [Live
messages](https://docs.browser-use.com/cloud/agent/streamin g) — stream the agent's messages alongside the live browser view
- [Follow-up
tasks](https://docs.browser-use.com/cloud/agent/follow-up-tasks) — chain tasks in one session while watching live

Playwright, Puppeteer, Selenium
Source:
https://docs.browser-use.com/cloud/browser/playwright-puppeteer-selenium

Every session runs in a [hardened Chromium fork](https://docs.browser-use.com/cloud/browser/stealth)

with stealth, anti-fingerprinting, and [residential proxies](https://docs.browser-use.com/cloud/browser/proxies) enabled by default — no configuration needed.

Option 1: WebSocket URL (no SDK)

Connect with a single URL. All configuration is passed as query parameters.

Playwright

` ` ` python Python
from playwright.async_api import async_playwright

WSS_URL = "wss://connect.browser-use.com? apiKey=YOUR_API_KEY&proxyCountryCode=us"

async with async_playwright() as p:
browser = await p.chromium.connect_over_cdp(WSS_URL) page = browser.contexts[0].pages[0]
await page.goto("https://example.com")
print(await page.title())
await browser.close()
Browser is automatically stopped when the WebSocket disconnects
` ` `

` ` ` typescript TypeScript
import { chromium } from "playwright";

const WSS_URL = "wss://connect.browser-use.com? apiKey=YOUR_API_KEY&proxyCountryCode=us";

const browser = await chromium.connectOverCDP(WSS_URL); const page = browser.contexts()[0].pages()[0];
await page.goto("https://example.com"); console.log(await page.title());
await browser.close();
// Browser is automatically stopped when the WebSocket disconnects
` ` `

Puppeteer

` ` ` typescript
import puppeteer from "puppeteer-core";

const WSS_URL = "wss://connect.browser-use.com? apiKey=YOUR_API_KEY&proxyCountryCode=us";

const browser = await
puppeteer.connect({ browserWSEndpoint: WSS_URL }); const [page] = await browser.pages();
await page.goto("https://example.com"); console.log(await page.title());
await browser.close();

` ` `

Selenium

Selenium requires a local WebSocket proxy to connect to Browser Use's remote CDP endpoint. Use [selenium-wire] (https://github.com/wkeeling/selenium-wire) or connect through Playwright's CDP bridge instead:

` ` ` python
from playwright.sync_api import sync_playwright

WSS_URL = "wss://connect.browser-use.com? apiKey=YOUR_API_KEY&proxyCountryCode=us"

with sync_playwright() as p:
browser = p.chromium.connect_over_cdp(WSS_URL) page = browser.contexts[0].pages[0] page.goto("https://example.com")
print(page.title())
browser.close()

` ` `

Selenium's `debugger_address` only supports local
` host:port` connections. For remote CDP over WebSocket, use
Playwright or Puppeteer instead.

Query parameters

| Parameter | Type | Description | |-----------|------|-------------|
| `apiKey` | `string` | **Required.** Your Browser Use API key. |
| `proxyCountryCode` | `string` | Proxy country code (e.g.
` us`, `de`, `jp`). 195+ countries. |
| `profileId` | `string` | Load a saved browser profile (cookies, localStorage). |
| `timeout` | `int` | Session timeout in minutes. Default:
15. Max: 240 (4 hours). |
| `browserScreenWidth` | `int` | Browser width in pixels. | | `browserScreenHeight` | `int` | Browser height in pixels. |

Option 2: SDK

Create a browser via the SDK, get a `cdp_url`, and connect with Playwright or Puppeteer.

Playwright

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse
from playwright.async_api import async_playwright

client = AsyncBrowserUse()
browser = await client.browsers.create() print(browser.cdp_url) # https://uuid.cdpN.browser-
use.com

print(browser.live_url) # https://live.browser-use.com?
wss=...

async with async_playwright() as p:
pw_browser = await
p.chromium.connect_over_cdp(browser.cdp_url)
page = pw_browser.contexts[0].pages[0]
await page.goto("https://example.com") print(await page.title())
await pw_browser.close()

await client.browsers.stop(browser.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";
import { chromium } from "playwright";

const client = new BrowserUse();
const browser = await client.browsers.create(); console.log(browser.cdpUrl); // https://uuid.cdpN.browser-
use.com

console.log(browser.liveUrl); // https://live.browser-use.com?wss=...

const pwBrowser = await chromium.connectOverCDP(browser.cdpUrl);
const page = pwBrowser.contexts()[0].pages()[0]; await page.goto("https://example.com"); console.log(await page.title());
await pwBrowser.close();

await client.browsers.stop(browser.id);

` ` `

Puppeteer

` ` ` typescript
import { BrowserUse } from "browser-use-sdk/v3";
import puppeteer from "puppeteer-core";

const client = new BrowserUse();
const browser = await client.browsers.create();

// Puppeteer needs the WebSocket URL from /json/version const resp = await fetch(`${browser.cdpUrl}/json/version`); const { webSocketDebuggerUrl } = await resp.json();

const pwBrowser = await
puppeteer.connect({ browserWSEndpoint: webSocketDebuggerUrl });
const [page] = await pwBrowser.pages();
await page.goto("https://example.com");
console.log(await page.title());
await pwBrowser.close();

await client.browsers.stop(browser.id);

` ` `

Always stop browser sessions when done. Sessions left running will continue to incur charges until the timeout expires.

Profiles
Source:
https://docs.browser-use.com/cloud/guides/authentication

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
profile = await client.profiles.create(name="user-id-1")
# or search existing
# profile = (await client.profiles.list(query="user-id-1")).items[0]
session = await
client.sessions.create(profile_id=profile.id)
result = await client.run("Check browser-use github stars", session_id=session.id)
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const profile = await client.profiles.create({ name: "user-id-1" });
// or search existing
// const profile = (await client.profiles.list({ query: "user-id-1" })).items[0];
const session = await client.sessions.create({ profileId: profile.id });
const result = await client.run("Check browser-use github stars", {
sessionId: session.id,
});

console.log(result.output);

` ` `

View your profile IDs at [cloud.browser-use.com/settings] (https://cloud.browser-use.com/settings?tab=profiles).

Manage profiles

` ` ` python Python
# Create
profile = await client.profiles.create(name="work-account")

# List all
response = await client.profiles.list()
for p in response.items:
print(p.id, p.name)

# Search by name
response = await client.profiles.list(query="user-id-1") profile = response.items[0] # first match

# Get one by ID
profile = await client.profiles.get(profile_id)

Update
await client.profiles.update(profile_id, name="renamed")

Delete
await client.profiles.delete(profile_id)

` ` `

` ` ` typescript TypeScript
// Create
const profile = await client.profiles.create({ name: "work-account" });

// List all
const response = await client.profiles.list();
for (const p of response.items) {
console.log(p.id, p.name);
}

// Search by name

const results = await client.profiles.list({ query: "user-id-1" });
const found = results.items[0]; // first match

// Get one by ID
const fetched = await client.profiles.get(profileId);

// Update
await client.profiles.update(profileId, { name: "renamed" });

// Delete
await client.profiles.delete(profileId);

` ` `

Usage patterns

- **Per-user profiles:** Create one profile per end-user. Query by name to get the profile ID, or store a mapping between your users and their profile IDs in your database.

Profile state is only saved when the session ends. Always call `sessions.stop()` when you are done — if a session is left open or times out, changes may not be persisted.

Sync local and cloud cookies
Source: https://docs.browser-use.com/cloud/guides/profile-sync

` ` ` bash
export BROWSER_USE_API_KEY=your_key && curl -fsSL https://browser-use.com/profile.sh | sh

` ` `

This opens a browser where you select which accounts to sync. After syncing, you receive a `profile_id` to use in your tasks.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
session = await
client.sessions.create(profile_id="your_synced_profile_id") result = await client.run("Check my LinkedIn messages", session_id=session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();
const session = await client.sessions.create({ profileId: "your_synced_profile_id" });
const result = await client.run("Check my LinkedIn messages", {
sessionId: session.id,
});

` ` `

2FA

Source: https://docs.browser-use.com/cloud/guides/2fa

Sites with 2FA block automated logins. Here are four approaches — pick the one that fits your setup.

Approach	Best for	Complexity
[Profiles (login once)](#1-profiles--login-once-reuse-cookies)	Sites with long-lived cookies	Lowest
[Human in the loop](#2-human-in-the-loop)	One-off tasks, complex auth flows	Low
[Agent Mail](#3-agent-mail)	Email-based 2FA, end-client automation	Medium
[TOTP secret in prompt](#4-totp-secret-in-prompt)	Authenticator app 2FA (Google Authenticator, Authy)	Medium

- - -

1. Profiles — login once, reuse cookies

Login manually once (or let the agent do it), then save the browser state as a profile. Future sessions reuse the cookies — no 2FA prompt as long as the cookies are valid.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

# Create a profile and a session
profile = await client.profiles.create(name="my-account") session = await
client.sessions.create(profile_id=profile.id)
print(f"Live view: {session.live_url}")

# Option A: human logs in via live view
input("Log in and complete 2FA in the live view, then press Enter...")

Option B: let the agent log in
await client.run("Log in to example.com with user@example.com / password123", session_id=session.id)

Stop the session — this saves cookies to the profile await client.sessions.stop(session.id)

# Next time: reuse the profile, no 2FA needed
session = await
client.sessions.create(profile_id=profile.id)
result = await client.run("Go to example.com/dashboard and get my balance", session_id=session.id) print(result.output)
await client.sessions.stop(session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3"; import * as readline from "readline";

const client = new BrowserUse();

// Create a profile and a session

const profile = await client.profiles.create({ name: "my-account" });
const session = await client.sessions.create({ profileId: profile.id });
console.log(`Live view: ${session.liveUrl}`);

// Option A: human logs in via live view
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
await new Promise((resolve) => rl.question("Log in and complete 2FA in the live view, then press Enter...", resolve));
rl.close();

// Option B: let the agent log in
// await client.run("Log in to example.com with user@example.com / password123", { sessionId: session.id });

// Stop the session — this saves cookies to the profile await client.sessions.stop(session.id);

// Next time: reuse the profile, no 2FA needed
const newSession = await
client.sessions.create({ profileId: profile.id }); const result = await client.run("Go to example.com/dashboard and get my balance", { sessionId: newSession.id });
console.log(result.output);
await client.sessions.stop(newSession.id);

` ` `

Cookies expire. Some sites stay logged in for months, others expire daily. If your sessions start hitting login pages again, re-authenticate and save the profile.

Always call `sessions.stop()` after you're done — profile state is only saved when the session ends cleanly.

- - -

2. Human in the loop

Let the agent navigate to the login page, then a human takes over to complete 2FA via the live browser view. The agent continues after.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()
session = await client.sessions.create()
print(f"Live view: {session.live_url}")

# Agent navigates to login
result = await client.run(
"Go to example.com/login and enter username user@example.com and password mypassword, then stop before 2FA",
session_id=session.id,
)

# Human completes 2FA in the live view
input("Complete 2FA in the live view, then press Enter...")

# Agent continues
result = await client.run(
"You are now logged in. Go to the dashboard and export the monthly report",
session_id=session.id,
)
print(result.output)
await client.sessions.stop(session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3"; import * as readline from "readline";

const client = new BrowserUse();
const session = await client.sessions.create(); console.log(`Live view: ${session.liveUrl}`);

// Agent navigates to login
await client.run(

"Go to example.com/login and enter username user@example.com and password mypassword, then stop before 2FA",
{ sessionId: session.id },
);

// Human completes 2FA in the live view
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
await new Promise((resolve) => rl.question("Complete 2FA in the live view, then press Enter...", resolve));
rl.close();

// Agent continues
const result = await client.run(
"You are now logged in. Go to the dashboard and export
the monthly report",
{ sessionId: session.id },
);
console.log(result.output);
await client.sessions.stop(session.id);

` ` `

See [Human in the
loop](https://docs.browser-use.com/cloud/agent/human-in-the-loop) for more patterns.

- - -

3. Agent Mail

When 2FA sends a code via email, the agent can read it automatically using Agent Mail — a built-in email inbox for each session.

Agent Mail is **enabled by default** (`agentmail=True`). Each session gets a unique email address (`session.agentmail_email`). The agent can send and receive emails during the task.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

result = await client.run(
"""

1. Go to example.com/signup
2. Sign up with the agent's email address (use the email available to you)
3. Check your email inbox for the verification code
4. Enter the code on the website
5. Complete the registration
""",
agentmail=True, # default, shown for clarity
)
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();

const result = await client.run(
` 1. Go to example.com/signup
2. Sign up with the agent's email address (use the email available to you)
3. Check your email inbox for the verification code
4. Enter the code on the website

5. Complete the registration`,
{ agentmail: true }, // default, shown for clarity
);
console.log(result.output);

` ` `

For end-client automation

If you're automating on behalf of your users and they need to receive 2FA codes:

1. **Email forwarding:** Have your client set up an email forwarding rule — forward all emails from the service (e.g., `noreply@bank.com`) to a dedicated inbox (a Gmail address or an Agent Mail address).

2. **Give the agent access:** The agent reads the forwarded 2FA code from that inbox during the task.

This way, your client's real email stays private — the agent only sees the forwarded verification emails.

Connect external email via Composio

You can also give the agent access to an existing Gmail account using [Composio](https://composio.dev) in the Browser Use dashboard. Once connected, the agent can read emails directly from that account to retrieve 2FA codes.

- - -

4. TOTP secret in prompt

If the site uses an authenticator app (Google Authenticator, Authy, etc.), you can pass the TOTP secret to the agent. Our agent can execute Python code, so it uses the `pyotp` library to generate fresh 6-digit codes on the fly.

When you set up 2FA on a site, instead of only scanning the QR code, also copy the **secret key** (usually shown as "manual entry" or "can't scan the QR code?"). This is a long base32 string like `JBSWY3DPEHPK3PXP`.

` ` ` python Python
from browser_use_sdk.v3 import AsyncBrowserUse

client = AsyncBrowserUse()

# The TOTP secret from your authenticator setup — NOT the 6-digit code
totp_secret = "JBSWY3DPEHPK3PXP"

result = await client.run(
f"""
Log into example.com with username user@example.com and password mypassword.
When prompted for a 2FA code, generate one using pyotp:

import pyotp
totp = pyotp.TOTP("{totp_secret}") code = totp.now()

Enter the generated code.
""",
)
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk/v3";

const client = new BrowserUse();

// The TOTP secret from your authenticator setup — NOT the 6-digit code
const totpSecret = "JBSWY3DPEHPK3PXP";

const result = await client.run(
` Log into example.com with username user@example.com and password mypassword.
When prompted for a 2FA code, generate one using pyotp:

import pyotp
totp = pyotp.TOTP("${totpSecret}")
code = totp.now()

Enter the generated code.`,
);
console.log(result.output);

` ` `

This works because the Browser Use agent can execute Python code as part of its task. The agent runs
` pyotp.TOTP(secret).now()` to generate a time-based 6-digit
code, then types it into the 2FA field.

Where to find TOTP secrets

- **1Password**: Edit item → One-Time Password → Show secret

- **Google Authenticator**: During setup, click "Can't scan it?" to see the key
- **Authy**: Export via desktop app settings
- **Most sites**: Look for "manual entry" or "setup key" during 2FA enrollment

- - -

Which approach should I use?

Start with **Profiles** — log in once, reuse cookies. If cookies expire frequently, add **TOTP secret in prompt** for fully automated re-login.
Use **Profiles** with one profile per user. For initial login, use **Human in the loop** — your user logs in once via the live view, then the agent reuses the session. For email 2FA, set up **Agent Mail** with email forwarding from your user.
Use **Agent Mail** (enabled by default). For end-client scenarios, have them forward 2FA emails to a dedicated inbox.
Use **TOTP secret in prompt** — the agent generates codes via pyotp, no human intervention needed.

OpenClaw
Source:
https://docs.browser-use.com/cloud/tutorials/integrations/o penclaw

[OpenClaw](https://openclaw.ai) is a self-hosted gateway that connects chat apps like WhatsApp, Telegram, and Discord to AI coding agents. Add Browser Use and those agents get full browser automation — anti-detect profiles, CAPTCHA solving, residential proxies in 195+ countries, and stealth browsing out of the box.

Two ways to set it up: connect a Browser Use cloud browser to OpenClaw's native browser tool via CDP, or install the Browser Use CLI as a skill.

Option 1: Cloud Browser via CDP

OpenClaw has a built-in browser tool with its own CLI commands (`openclaw browser`). By default, it controls a local Chromium instance. You can point it at a Browser Use cloud browser instead by configuring a remote CDP profile.

Browser Use exposes a WebSocket CDP URL. OpenClaw connects to it like any remote browser — no SDK or extra dependencies needed.

Setup

1. Get your API key

Sign up at cloud.browser-use.com and copy your API key from [Settings → API Keys] (https://cloud.browser-use.com/settings?tab=api-keys&new=1).

2. Add a Browser Use profile

Open `~/.openclaw/openclaw.json` and add a `browser-use` profile:

` ` ` json5
{
browser: {
enabled: true,
defaultProfile: "browser-use", remoteCdpTimeoutMs: 3000, remoteCdpHandshakeTimeoutMs: 5000,
profiles: {
"browser-use": {
cdpUrl: "wss://connect.browser-use.com? apiKey=<BROWSER_USE_API_KEY>&proxyCountryCode=us", color: "#ff750e",
},
},
},
}

` ` `

Replace `<BROWSER_USE_API_KEY>` with your actual key. All Browser Use session parameters can be passed as query params in the `cdpUrl`:

- `timeout` — session duration in minutes (max 240)
- `profileId` — load a saved browser profile with persistent cookies and localStorage
- `proxyCountryCode` — route traffic through a specific country (e.g. `us`, `de`, `jp`)

3. Use it

OpenClaw's browser commands now run against a Browser Use cloud browser:

` ` ` bash
openclaw browser --browser-profile browser-use open https://example.com
openclaw browser --browser-profile browser-use snapshot openclaw browser --browser-profile browser-use screenshot

` ` `

If you set `defaultProfile` to `"browser-use"` in the config (as shown above), you can drop the `--browser-profile` flag:

` ` ` bash
openclaw browser open https://example.com
openclaw browser snapshot
openclaw browser screenshot

` ` `

Option 2: Browser Use CLI

The Browser Use CLI is a standalone tool that gives any OpenClaw agent browser automation through a SKILL.md file. The agent reads the skill and learns to use the CLI commands directly. It's available on [skills.sh](https://skills.sh/browser-use/browser-use/brows er-use) and [ClawHub](https://clawhub.ai/ShawnPana/browser-use).

Setup

1. Install the CLI

` ` ` bash
curl -fsSL https://browser-use.com/cli/install.sh | bash

` ` `

2. Verify the installation

` ` ` bash
browser-use doctor
` ` `

3. Install the skill

Ask your OpenClaw agent to install it directly from [ClawHub](https://clawhub.ai/ShawnPana/browser-use), or install from
[skills.sh](https://skills.sh/browser-use/browser-use/brows er-use):

` ` ` bash
npx skills add https://github.com/browser-use/browser-use
- - skill browser-use
` ` `

Once the skill is loaded, OpenClaw agents can use the
` browser-use` CLI to navigate pages, click elements, fill
forms, take screenshots, extract data, and more. The skill file teaches the agent the full command set.

For the complete CLI reference and advanced features like cloud browsers, tunnels, sessions, and Python execution, see the [README](https://github.com/browser-use/browser-use/blob/main/browser_use/skill_cli/README.md) and the [Browser Use docs](https://docs.browser-use.com).

MCP Server
Source: https://docs.browser-use.com/cloud/guides/mcp-

server

` ` `

https://api.browser-use.com/v3/mcp

` ` `

Get your API key at [cloud.browser-use.com/settings] (https://cloud.browser-use.com/settings?tab=api-keys&new=1).

Claude Code

` ` ` bash
claude mcp add -t http -H "x-browser-use-api-key: YOUR_API_KEY" browser-use https://api.browser-use.com/v3/mcp

` ` `

Claude Desktop

Add to `claude_desktop_config.json`:

` ` ` json
{
"mcpServers": {
"browser-use": {
"url": "https://api.browser-use.com/v3/mcp", "headers": {
"x-browser-use-api-key": "YOUR_API_KEY"
}
}
}
}

` ` `

Cursor

Add to `.cursor/mcp.json`:

` ` ` json
{

"mcpServers": {
"browser-use": {
"url": "https://api.browser-use.com/v3/mcp", "headers": {
"x-browser-use-api-key": "YOUR_API_KEY"
}
}
}
}

` ` `

Windsurf

Add to `~/.codeium/windsurf/mcp_config.json`:

` ` ` json
{
"mcpServers": {
"browser-use": {
"serverUrl": "https://api.browser-use.com/v3/mcp", "headers": {
"x-browser-use-api-key": "YOUR_API_KEY"
}
}
}
}

` ` `

Available Tools

Tool	Description
`run_session`	Create a session and run a task. Supports
` keep_alive`, `model` (`bu-mini`, `bu-max`),
` output_schema`, and `profile_id`.
`get_session`	Poll session status and output. Returns status, step count, cost breakdown, and live URL.
`send_task`	Send a follow-up task to an idle keep-alive session.
`stop_session`	Stop a session. `strategy: "task"` stops only the task, `"session"` destroys the sandbox.
`get_session_messages`	Get the agent's messages —

browser actions, reasoning, and results. |
| `list_sessions` | List recent sessions with status and cost. |
| `list_browser_profiles` | List browser profiles for authenticated tasks. |

Webhooks
Source: https://docs.browser-use.com/cloud/guides/webhooks

Set up webhooks at cloud.browser-use.com/settings? tab=webhooks.

Events

Event	When
`agent.task.status_update`	Task status changes (`running`, `idle`, or `stopped`)
`test`	Webhook test ping

Payload

` ` ` json
{
"type": "agent.task.status_update", "timestamp": "2025-01-15T10:30:00Z", "payload": {
"task_id": "task_abc123", "session_id": "session_xyz", "status": "idle",
"metadata": {}
}
}

` ` `

Signature verification

Every webhook request includes two headers:

- `X-Browser-Use-Signature` — HMAC-SHA256 signature of the payload
- `X-Browser-Use-Timestamp` — Unix timestamp (seconds) when the request was sent

The signature is computed over `{timestamp}.{body}`, where
` body` is the JSON-serialized payload with keys sorted
alphabetically and no extra whitespace. Verify it to ensure the request is authentic and to prevent replay attacks.

` ` ` python Python
import hashlib
import hmac
import json
import time

def verify_webhook(body: bytes, signature: str, timestamp: str, secret: str) -> bool:
# Reject requests older than 5 minutes
try:
ts = int(timestamp)
except (ValueError, TypeError):
return False
if abs(time.time() - ts) > 300:
return False
payload = json.loads(body)
message = f"{timestamp}.{json.dumps(payload, separators=(',', ':'), sort_keys=True)}"
expected = hmac.new(secret.encode(), message.encode(), hashlib.sha256).hexdigest()
return hmac.compare_digest(expected, signature)

` ` `

` ` ` typescript TypeScript
import { createHmac, timingSafeEqual } from "crypto";

function sortKeys(obj: unknown): unknown {
if (Array.isArray(obj)) return obj.map(sortKeys);
if (obj !== null && typeof obj === "object") {
return Object.keys(obj as object)
. sort()
. reduce((acc, key) => {
(acc as Record<string, unknown>)[key] = sortKeys((obj

as Record<string, unknown>)[key]);
return acc;
}, {} as Record<string, unknown>);
}
return obj;
}

function verifyWebhook(body: string, signature: string, timestamp: string, secret: string): boolean {
// Reject requests older than 5 minutes
if (Math.abs(Date.now() / 1000 - parseInt(timestamp)) >
300) return false;
const payload = JSON.parse(body);
const message = `${timestamp}.$ {JSON.stringify(sortKeys(payload))}`;
const expected = createHmac("sha256", secret).update(message).digest("hex");
return timingSafeEqual(Buffer.from(expected), Buffer.from(signature));
}

` ` `

Example: Express webhook handler

` ` ` typescript
import express from "express";
import { createHmac, timingSafeEqual } from "crypto";

const app = express();
app.use(express.raw({ type: "application/json" }));

const WEBHOOK_SECRET = process.env.WEBHOOK_SECRET!;

function sortKeys(obj: unknown): unknown {
if (Array.isArray(obj)) return obj.map(sortKeys);
if (obj !== null && typeof obj === "object") {
return Object.keys(obj as object)
. sort()
. reduce((acc, key) => {
(acc as Record<string, unknown>)[key] = sortKeys((obj
as Record<string, unknown>)[key]);
return acc;

}, {} as Record<string, unknown>);
}
return obj;
}

app.post("/webhook", (req, res) => {
const signature = req.headers["x-browser-use-signature"] as string;
const timestamp = req.headers["x-browser-use-timestamp"] as string;

if (Math.abs(Date.now() / 1000 - parseInt(timestamp)) >
300) {
return res.status(401).send("Request too old");
}

const body = req.body.toString();
const payload = JSON.parse(body);
const message = `${timestamp}.$ {JSON.stringify(sortKeys(payload))}`;
const expected = createHmac("sha256", WEBHOOK_SECRET).update(message).digest("hex");

if (!timingSafeEqual(Buffer.from(expected), Buffer.from(signature))) {
return res.status(401).send("Invalid signature");
}

if (payload.type === "agent.task.status_update") { const { task_id, status, session_id } = payload.payload; console.log(`Task ${task_id} is now ${status}`);
}

res.status(200).send("OK");
});

app.listen(3000);

` ` `

Example: FastAPI webhook handler

` ` ` python

from fastapi import FastAPI, Request, HTTPException import hashlib
import hmac
import json
import os
import time

app = FastAPI()

WEBHOOK_SECRET = os.environ["WEBHOOK_SECRET"]

@app.post("/webhook")
async def handle_webhook(request: Request):
body = await request.body()
signature = request.headers.get("x-browser-use-signature", "")
timestamp = request.headers.get("x-browser-use-timestamp", "")

# Reject requests older than 5 minutes
try:
ts = int(timestamp)
except (ValueError, TypeError):
raise HTTPException(status_code=401, detail="Invalid timestamp")
if abs(time.time() - ts) > 300:
raise HTTPException(status_code=401, detail="Request too old")

payload = json.loads(body)
message = f"{timestamp}.{json.dumps(payload, separators=(',', ':'), sort_keys=True)}"
expected = hmac.new(WEBHOOK_SECRET.encode(), message.encode(), hashlib.sha256).hexdigest()

if not hmac.compare_digest(expected, signature):
raise HTTPException(status_code=401, detail="Invalid signature")

if payload["type"] == "agent.task.status_update": task_id = payload["payload"]["task_id"]
status = payload["payload"]["status"]

print(f"Task {task_id} is now {status}")

return {"status": "ok"}

` ` `

For local development, use a tunneling tool like [ngrok] (https://ngrok.com) to expose your local server: `ngrok http 3000`. Then set the ngrok URL as your webhook endpoint in the dashboard.

n8n
Source:
https://docs.browser-use.com/cloud/tutorials/integrations/n 8n

Browser Use works with [n8n](https://n8n.io) as a standard HTTP integration — no custom nodes needed.

1. Create a credential

In n8n, go to **Credentials → Add Credential → Header Auth** and set:

Field	Value
Name	`Authorization`
Value	`Bearer YOUR_API_KEY`

Get your API key at [cloud.browser-use.com/settings] (https://cloud.browser-use.com/settings?tab=api-keys&new=1).

2. Start a session

Add an **HTTP Request** node:

Setting	Value
Method	`POST`
URL	`https://api.browser-use.com/api/v3/sessions`

| Authentication | Header Auth (from step 1) |
| Body Type | JSON |

Body:
` ` ` json
{
"task": "Find the top 3 trending repos on GitHub today"
}

` ` `

The response includes a `session_id` you'll use to poll for results.

3. Poll for completion

Add a second **HTTP Request** node in a loop:

Setting	Value
Method	`GET`
URL
` https://api.browser-use.com/api/v3/sessions/{{ $json.id }} `
Authentication	Header Auth (from step 1)

Check the `status` field. The session is done when status is `idle`, `stopped`, `error`, or `timed_out`. Use an **If** node to loop back with a **Wait** node (5–10 seconds) until complete.

The final response contains `output` with the agent's result.

Event-driven alternative

Instead of polling, use [Webhooks](https://docs.browser-use.com/cloud/guides/webhooks) to receive a callback when the session completes. Configure your webhook endpoint in the [dashboard](https://cloud.browser-use.com/settings? tab=webhooks), then add a **Webhook** trigger node in n8n to receive `agent.task.status_update` events when sessions finish.

This pattern works with any workflow tool that supports HTTP requests — Make, Zapier, Pipedream, or custom orchestrators.

Chat UI
Source: https://docs.browser-use.com/cloud/tutorials/chat-ui

Clone and run in minutes. Next.js + Browser Use SDK v3.

This tutorial walks through the [chat-ui-example](https://github.com/browser-use/chat-ui-example) — a Next.js app that lets users chat with a Browser Use agent in real time. We focus on the SDK integration, not the UI components.

The app has two pages:

1. **Home** — the user types a task, the app creates a session and sends the task.
2. **Session** — live browser preview, streaming messages, follow-ups, and recording download.

All SDK calls live in a single file: `src/lib/api.ts`.

Setup

` ` ` typescript api.ts
import { BrowserUse } from "browser-use-sdk/v3";

// Server-only — no NEXT_PUBLIC_ prefix, never exposed to the browser
const apiKey = process.env.BROWSER_USE_API_KEY ?? ""; export const client = new BrowserUse({ apiKey });

` ` `

The API key uses `BROWSER_USE_API_KEY` (no `NEXT_PUBLIC_` prefix) so it stays server-side. All SDK calls go through [server actions](https://nextjs.org/docs/app/guides/forms)

— never call the SDK directly from client components.

- - -

1. Create a session

` ` ` typescript actions.ts
"use server";
import { client } from "./api";

export async function createSession() {
const session = await client.sessions.create({ keepAlive: true,
enableRecording: true,
});
return { id: session.id, liveUrl: session.liveUrl,
status: session.status };
}

` ` `

- **`keepAlive: true`** keeps the session open after each task so the user can send follow-ups (default is `false`).
- **`enableRecording: true`** produces an MP4 video of the browser session.

- **`liveUrl`** is returned immediately — no waiting or extra call needed.

The home page creates the session, navigates to the session page (passing `liveUrl` and the initial task via URL params), and the session page takes over from there:

` ` ` typescript page.tsx
async function handleSend(message: string) {
const session = await createSession();

router.push(
` /session/${session.id}?liveUrl=$
{encodeURIComponent(session.liveUrl)}&task=$ {encodeURIComponent(message)}`
);
}

` ` `

- - -

2. Stream messages with `for await`

Instead of polling `sessions.get()` and
` sessions.messages()` separately, use `client.run()` — it
streams messages and resolves when the task completes:

` ` ` typescript session-context.tsx
const streamTask = useCallback(async (task: string) => { const run = client.run(task, { sessionId });

for await (const msg of run) {
setMessages((prev) => [...prev, msg]);
}

// Iterator done — task reached terminal state setSession(run.result);
}, [sessionId]);

` ` `

The `for await` loop yields each message as it arrives. When the loop ends, `run.result` contains the final session state (status, output, etc.). No separate status polling needed.

Wire it up in a `useEffect` to auto-run the initial task from URL params:

` ` ` typescript session-context.tsx
useEffect(() => {
if (!initialTask) return;
sendMessage(initialTask);
}, []);

` ` `

- - -

3. Follow-up tasks

Follow-ups call the same `streamTask` function — the stream

already includes the user message, so no optimistic insert is needed:

` ` ` typescript session-context.tsx
const sendMessage = useCallback(async (task: string) => { await streamTask(task);
}, [streamTask]);

` ` `

The SDK auto-sets `keepAlive: true` when targeting an existing session, so follow-up tasks work without extra config.

- - -

4. Recording

Fetch the MP4 URL after the session ends (recording was enabled in step 1):

` ` ` typescript session-context.tsx
useEffect(() => {
if (!isTerminal) return;

client.sessions.waitForRecording(sessionId).then((urls) => {
if (urls.length) setRecordingUrls(urls);
});
}, [isTerminal, sessionId]);

` ` `

`waitForRecording` polls for up to 15 seconds and returns presigned MP4 download URLs. Returns an empty array if the agent answered without opening a browser.

- - -

5. Stop a task

` ` ` typescript actions.ts
export async function stopTask(id: string) {
await client.sessions.stop(id, { strategy: "task" });

}

` ` `

Using `strategy: "task"` stops only the current task, keeping the session alive for follow-ups.

- - -

6. Session page

The session page consumes everything through a context provider:

` ` ` typescript session/[id]/page.tsx
function SessionPage() {
const { session, turns, isBusy, isTerminal, recordingUrls, sendMessage, stopTask } = useSession();

return (
 {/* Chat column */}
 <ChatMessages turns={turns} isBusy={isBusy} /> <ChatInput
onSend={sendMessage}
onStop={stopTask}
disabled={isTerminal}
/>

{/* Live browser view — liveUrl available from session creation */}
<BrowserPanel liveUrl={session?.liveUrl} />

);
}

` ` `

- - -

Summary

Method	Purpose
`client.sessions.create()`	Create a session (returns
` liveUrl` immediately)
`client.run()`	Send a task and stream messages with
` for await`
`client.sessions.stop()`	Stop the current task
`client.sessions.waitForRecording()`	Get MP4 recording URLs

FAQ
Source: https://docs.browser-use.com/cloud/faq

Which model should I use?

- **bu-ultra** — most capable. Use for the hardest tasks that need maximum accuracy.
- **bu-max** (default) — best balance of capability and cost. Use for complex multi-step workflows.
- **bu-mini** — faster, cheaper. Good for simple tasks.

How do I get the live browser URL?

` live_url` is returned on session creation. Embed it in an
iframe or open it in a browser.

` ` ` python
session = await client.sessions.create(task="Go to example.com")
print(session.live_url)

` ` `

Getting blocked by a website

Stealth and proxies are active by default. If you're still getting blocked:

- **Use a profile** with logged-in cookies to bypass login walls.

- **Try a different proxy country** to match the target region.

If it still doesn't work, contact support inside the [Cloud Dashboard](https://cloud.browser-use.com) — send us a link to the page where you're getting blocked.

Rate limited (429 errors)

The SDK auto-retries 429 responses with exponential backoff. If persistent, you may need more concurrent sessions — contact support.

v2 vs v3 — which should I use?

v3 is the recommendation for everything. It's a premium agent (not available in open source) that is significantly more capable than v2:

- **Much better at complex tasks** and multi-step workflows
- **Much better at large data extraction**
- **File system** with persistent memory across tasks
- **Task scheduling** with 1,000+ integrations (Gmail, Slack, and more)

v2 is the closest to the open-source experience — pure browser automation, nothing else. If the open source already works great for your use case, v2 is the natural fit. For everything else, use v3.

` ` ` python
# v3 (recommended)
from browser_use_sdk.v3 import AsyncBrowserUse

# v2 (simple browser-only tasks)
from browser_use_sdk.v2 import AsyncBrowserUse

` ` `

Agent (v2)
Source: https://docs.browser-use.com/cloud/legacy/agent

Models

Model	API String	Cost per Step
Browser Use 2.0 (default)	`browser-use-2.0`	\$0.006		Browser Use LLM	`browser-use-llm`	\$0.002
O3	`o3`	\$0.03
Gemini 3 Pro Preview	`gemini-3-pro-preview`	\$0.03		Gemini 3 Flash Preview	`gemini-3-flash-preview`	\ $0.01
Gemini Flash Latest	`gemini-flash-latest`	\$0.0075		Gemini Flash Lite Latest	`gemini-flash-lite-latest`	\ $0.005
Claude Sonnet 4.5	`claude-sonnet-4-5-20250929`	\$0.05
Claude Sonnet 4.6	`claude-sonnet-4-6`	\$0.05

Pass `llm` explicitly to select a model:

` ` ` python Python
result = await client.run("...", llm="browser-use-2.0")

` ` `

` ` ` typescript TypeScript
const result = await client.run("...", { llm: "browser-use-2.0" });

` ` `

- - -

Files

Upload images, PDFs, documents, and text files (10 MB max) to sessions, and download output files from completed tasks.

Upload a file

Get a presigned URL, then upload via POST.

` ` ` python Python
import httpx

from browser_use_sdk import AsyncBrowserUse

client = AsyncBrowserUse()
session = await client.sessions.create()

upload = await client.files.session_url(session.id,
file_name="input.pdf", content_type="application/pdf",
size_bytes=1024,
)

with open("input.pdf", "rb") as f:
async with httpx.AsyncClient() as http:
await http.post(upload.url, content=f.read(), headers={"Content-Type": "application/pdf"})

result = await client.run("Summarize the uploaded PDF", session_id=session.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk";
import { readFileSync } from "fs";

const client = new BrowserUse();
const session = await client.sessions.create();

const upload = await client.files.sessionUrl(session.id, { fileName: "input.pdf",
contentType: "application/pdf",
sizeBytes: 1024,
});

await fetch(upload.url, {
method: "POST",
body: readFileSync("input.pdf"),
headers: { "Content-Type": "application/pdf" },
});

const result = await client.run("Summarize the uploaded PDF", { sessionId: session.id });

` ` `

Presigned URLs expire after 120 seconds. Max file size: 10 MB.

Download task output files

` ` ` python Python
result = await client.tasks.get(task_id)
for file in result.output_files:
output = await client.files.task_output(task_id, file.id) print(output.download_url) # download URL

` ` `

` ` ` typescript TypeScript
const result = await client.tasks.get(taskId);
for (const file of result.outputFiles) {
const output = await client.files.taskOutput(taskId, file.id);
console.log(output.downloadUrl);
}

` ` `

- - -

Streaming steps

Use `async for` to yield steps as the agent works.

` ` ` python Python
from browser_use_sdk import AsyncBrowserUse

client = AsyncBrowserUse()
run = client.run("Find the most upvoted post on Reddit r/technology today")
async for step in run:
print(f"Step {step.number}: {step.next_goal}")
print(f" URL: {step.url}")

print(run.result.output) # final result after iteration

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk";

const client = new BrowserUse();
const run = client.run("Find the most upvoted post on Reddit r/technology today");
for await (const step of run) {
console.log(`Step ${step.number}: ${step.nextGoal}`); console.log(` URL: ${step.url}`);
}

console.log(run.result?.output); // final result after iteration
` ` `

- - -

Key parameters

Parameter	Type	Description		-----------	------	-------------
`task`	`str`	What you want the agent to do. 1-50,000 characters.
`llm`	`str`	Model override. Default: Browser Use 2.0.
`output_schema` / `schema`	Pydantic / Zod	Schema for structured output.
`session_id`	`str`	Reuse an existing session. Omit for auto-session.
`start_url`	`str`	Initial page URL. Saves steps — send the agent directly there.
`secrets`	`dict`	Domain-specific credentials. See [Authentication](https://docs.browser-use.com/cloud/guides/authentication).
`allowed_domains`	`list[str]`	Restrict agent to these domains only.
`session_settings`	`SessionSettings`	Proxy, profile, browser config. See [Profiles](https://docs.browser-use.com/cloud/guides/authentication).
`flash_mode`	`bool`	Faster but less careful navigation.
`thinking`	`bool`	Enable extended reasoning.
`vision`	`bool | str`	Enable screenshots for the agent.
`highlight_elements`	`bool`	Highlight interactive

elements on the page. |
`system_prompt_extension`	`str`	Append custom instructions to the system prompt.
`judge`	`bool`	Enable quality judge to verify output.
`skill_ids`	`list[str]`	Skills the agent can use during the task.
`op_vault_id`	`str`	1Password vault ID for auto-fill credentials and 2FA.
`metadata`	`dict[str, str]`	Custom metadata attached to the task.

Public share links (v2)
Source: https://docs.browser-use.com/cloud/legacy/public-share

Generate a public URL that anyone can open to watch the entire agent session — no API key needed. Useful for sharing with teammates, stakeholders, or embedding in dashboards.

` ` ` python Python
share = await client.sessions.create_share(session.id) print(share.share_url)

` ` `

` ` ` typescript TypeScript
const share = await
client.sessions.createShare(session.id); console.log(share.shareUrl);

` ` `

Skills
Source: https://docs.browser-use.com/cloud/legacy/skills

A skill turns a website interaction into a reusable, reliable API. Describe what you need, Browser Use builds the automation, you call it like a function.

How skills work

Every skill has two parts:

- **Goal** — the full specification: what parameters it accepts, what data it returns, and the complete scope of work. If you want to extract data from 100 listings, the goal describes extracting from *all* of them.

- **Demonstration** (`agent_prompt`) — shows *how* to perform the task, but only once. Think of it like onboarding a new colleague: you would not walk them through all 100 listings. You would show the first one or two and say "keep going like this for the rest." The demonstration navigates the site, triggers the necessary network requests, and the system builds the actual endpoint logic from that recording.

The demonstration only needs to trigger the right network requests — it does not need to complete the full task. If extracting from many pages, open the first item and maybe paginate once. The system handles the rest.

Create a skill

` ` ` python Python
from browser_use_sdk import AsyncBrowserUse

client = AsyncBrowserUse()
skill = await client.skills.create(
goal="Extract the top X posts from HackerNews. For each post return: title, URL, score, author, comment count, and rank. X is an input parameter.",
agent_prompt="Go to https://news.ycombinator.com, click on the first post to load its content, go back to the list, and scroll down to trigger loading of additional posts.",)
print(skill.id)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk";

const client = new BrowserUse();
const skill = await client.skills.create({
goal: "Extract the top X posts from HackerNews. For each post return: title, URL, score, author, comment count, and rank. X is an input parameter.",
agentPrompt: "Go to https://news.ycombinator.com, click on the first post to load its content, go back to the list, and scroll down to trigger loading of additional posts.", });
console.log(skill.id);

` ` `

Skill creation takes ~30 seconds. You can also create skills visually from the [Cloud Dashboard](https://cloud.browser-use.com/skills) — record yourself performing the task, or let the agent demonstrate it.

Execute a skill

` ` ` python Python
result = await client.skills.execute(
skill.id,
parameters={"X": 10},
)
print(result)

` ` `

` ` ` typescript TypeScript
const result = await client.skills.execute(skill.id, { parameters: { X: 10 },
});
console.log(result);

` ` `

Refine with feedback

If execution is not quite right, iterate for free:

` ` ` python Python
await client.skills.refine(skill.id, feedback="Also extract the product description and available colors")

` ` `

` ` ` typescript TypeScript
await client.skills.refine(skill.id, {
feedback: "Also extract the product description and available colors",
});

` ` `

Marketplace

Browse and use community-created skills.

` ` ` python Python
skills = await client.marketplace.list()
my_skill = await client.marketplace.clone(skill_id) result = await client.marketplace.execute(skill_id, parameters={...})

` ` `

` ` ` typescript TypeScript
const skills = await client.marketplace.list();
const mySkill = await client.marketplace.clone(skillId); const result = await client.marketplace.execute(skillId,
{ parameters: { ... } });

` ` `

See [Pricing](https://browser-use.com/pricing) for skill costs.

1Password & 2FA
Source: https://docs.browser-use.com/cloud/guides/1password

Setup

1. Create a dedicated vault

Create a new vault in 1Password for Browser Use. Add the credentials you want the agent to access (usernames, passwords, and 2FA/TOTP codes).

2. Create a service account token

1. Go to [1Password Developer Tools - Service Accounts] (https://my.1password.eu/developer-tools/active/service-accounts)
2. Click **New Service Account**, name it "Browser Use Cloud"
3. Grant **read access** to the dedicated vault
4. Copy the generated token

3. Connect to Browser Use Cloud

1. Go to [Browser Use Cloud Settings - Secrets](https://cloud.browser-use.com/settings? tab=secrets)
2. Click **Create Integration**
3. Paste your service account token

Run tasks with 1Password

` ` ` python Python
from browser_use_sdk import AsyncBrowserUse

client = AsyncBrowserUse()
result = await client.run(
"Log into my Jira account and create a new ticket", op_vault_id="your-vault-id", allowed_domains=["*.atlassian.net"],
)
print(result.output)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk";

const client = new BrowserUse();
const result = await client.run(
"Log into my Jira account and create a new ticket", {
opVaultId: "your-vault-id",
allowedDomains: ["*.atlassian.net"],
},
);
console.log(result.output);

` ` `

For SSO/OAuth redirects, include all required domains:

` ` ` python Python
result = await client.run(
"Log into Jira and create a ticket for the Q4 release", op_vault_id="your-vault-id", allowed_domains=["*.atlassian.net", "*.okta.com"],
)

` ` `

` ` ` typescript TypeScript
const result = await client.run(
"Log into Jira and create a ticket for the Q4 release", {
opVaultId: "your-vault-id",
allowedDomains: ["*.atlassian.net", "*.okta.com"],
},
);

` ` `

How it works

When the agent encounters a login form:

1. It identifies the service (e.g., Twitter, GitHub, LinkedIn)
2. Retrieves matching credentials from your 1Password vault
3. Fills in the username and password
4. If 2FA is required and a TOTP code is stored, it generates and enters the code automatically

The agent never sees your actual credentials. The actual username, password, and 2FA codes are filled in programmatically — keeping your secrets hidden from the AI model.

Secrets
Source: https://docs.browser-use.com/cloud/guides/secrets

Pass credentials to the agent scoped by domain. The agent

uses them only on matching domains.

` ` ` python Python
from browser_use_sdk import AsyncBrowserUse

client = AsyncBrowserUse()
result = await client.run(
"Log into GitHub and star the browser-use/browser-use repo",
secrets={"github.com": "username:password123"}, allowed_domains=["github.com"],
)

` ` `

` ` ` typescript TypeScript
import { BrowserUse } from "browser-use-sdk";

const client = new BrowserUse();
const result = await client.run(
"Log into GitHub and star the browser-use/browser-use repo",
{
secrets: { "github.com": "username:password123" }, allowedDomains: ["github.com"],
},
);

` ` `

Use `allowed_domains` to restrict the agent to specific domains. Supports wildcards: `example.com`,
` *.example.com`.

For SSO/OAuth redirects, include all domains in the auth flow:

` ` ` python Python
result = await client.run(
"Log into the company portal and download the Q4 report", secrets={
"portal.example.com": "user@company.com:password123", "okta.com": "user@company.com:password123",
},
allowed_domains=["portal.example.com", "*.okta.com"],

)

` ` `

` ` ` typescript TypeScript
const result = await client.run(
"Log into the company portal and download the Q4 report", {
secrets: {
"portal.example.com": "user@company.com:password123", "okta.com": "user@company.com:password123",
},
allowedDomains: ["portal.example.com", "*.okta.com"],
},
);

` ` `

API Reference
Source: https://docs.browser-use.com/cloud/api-reference

Authentication

All requests require an API key in the `X-Browser-Use-API-Key` header:

` ` `

X-Browser-Use-API-Key: bu_your_key_here

` ` `

Get a key at
[cloud.browser-use.com/settings](https://cloud.browser-use.com/settings?tab=api-keys&new=1). Keys start with
` bu_`.

Base URL

` ` `

https://api.browser-use.com/api/v3

` ` `

Quick example

` ` ` bash Create a session
curl -X POST https://api.browser-use.com/api/v3/sessions \
- H "X-Browser-Use-API-Key: bu_your_key_here" \
- H "Content-Type: application/json" \
- d '{"task": "Find the top 3 trending repos on GitHub
today"}'

` ` `

` ` ` bash Get session result (replace SESSION_ID)
curl https://api.browser-use.com/api/v3/sessions/SESSION_ID \
- H "X-Browser-Use-API-Key: bu_your_key_here"

` ` `

Environment variable

Set the key once so SDKs pick it up automatically:

` ` ` bash
export BROWSER_USE_API_KEY=bu_your_key_here

` ` `

- - -

Prefer the SDK? See the [Agent docs](https://docs.browser-use.com/cloud/agent/quickstart) — the SDK has all API endpoints available as methods, including
` client.browsers.create()`.

` ` ` bash Python
pip install browser-use-sdk

` ` `

` ` ` bash TypeScript
npm install browser-use-sdk

` ` `

API key
Source: https://docs.browser-use.com/cloud/api-v2-overview

Get a key at

[cloud.browser-use.com/settings](https://cloud.browser-use.com/settings?tab=api-keys&new=1), then:

` ` ` bash
export BROWSER_USE_API_KEY=your_key

` ` `

Base URL: `https://api.browser-use.com/api/v2`

- - -

Prefer the SDK? See the [Agent (v2) docs](https://docs.browser-use.com/cloud/legacy/agent).

` ` ` bash Python
pip install browser-use-sdk

` ` `

` ` ` bash TypeScript
npm install browser-use-sdk

` ` `